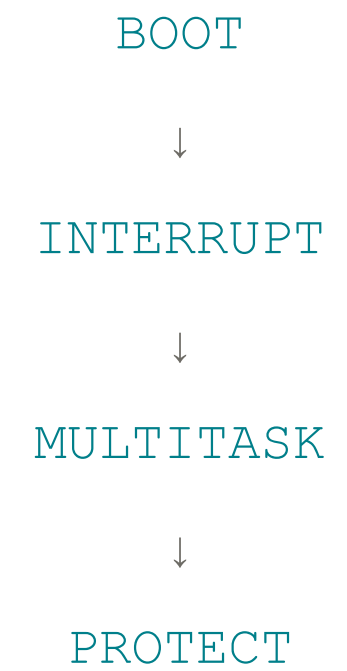


Operating-System Operations

Section 1.4 — Booting, interrupts, multiprogramming, dual-mode operation, and the timer



Prof. Mostafa Elhosseini

Professor of Smart Systems Engineering

Operating-System Operations

01

Boot

How the machine starts

02

Respond

How events reach the OS

03

Share

How programs share the CPU

04

Protect

How the OS stays safe

Today's Lecture

Part 1

Booting the system

Bootstrap, kernel, system programs, daemons, **systemd**.

Part 2

Interrupts, traps, exceptions

How the OS learns that something happened — and **system calls**.

Part 3

Multiprogramming and multitasking

Processes, CPU utilization, memory management, virtual memory.

Part 4

Protection: modes and the timer

Mode bit, privileged instructions, multimode CPUs, timer interrupts.

The Bootstrap Program

An operating system provides the **environment within which programs are executed**. For a computer to start running, it needs an initial program: the **bootstrap program**.

Stored in firmware

Held **within the computer hardware**, so it is available the instant power is applied.

ROM / EEPROM / Flash memory

Initializes everything

CPU registers, device controllers, and memory contents — all aspects of the system.

Wake up the hardware and make it ready.

Loads and starts the OS

It must know how to **locate the kernel, load it into memory, and begin executing it.**

Windows Boot Manager
GRUB on Linux

- The CPU cannot simply execute it directly from disk as normal working code. The kernel must be copied into RAM.
- So: **SSD/HDD → Kernel → RAM**

From Kernel to Services: systemd and Daemons

Kernel starts → first system program starts → system services/daemons (الشياطين) start → machine becomes ready for normal use

Not all services come directly from the kernel itself. Some are provided by **system programs** , loaded into memory during boot.

These become **daemons** : background processes (تعمل في الخلفية) that keep running **as long as the kernel is active** .

systemd

The **first system program** that starts running right after the Linux kernel finishes booting - It normally becomes PID 1.

It is an **init system and service manager** — it starts many other daemons. Once this phase completes, the system is **fully booted and waits for an event** .

Print daemon

→ waits → user sends print job → daemon wakes up → sends job to printer → waits again.

From Kernel to Services: systemd and Daemons

Kernel starts → first system program starts → system services/daemons (الشياطين) start → machine becomes ready for normal use

- Not all services come directly from the kernel itself.
- Putting everything inside the kernel would make it **very large, complex, and risky**.

systemd

The **first system program** that starts running right after the Linux kernel finishes booting - It normally becomes PID 1.

systemd acts almost like a manager of background services.

- **Analogy: A university opening in the morning.**
Kernel = building infrastructure
Electricity, rooms, doors, basic facilities.

systemd = operations manager

It tells each department:

Security, start.

Network services, start.

Logging service, start.

Login service, start.

Then each department continues operating in the background.

Two Families of Interrupt

How the operating system learns that something has happened — from hardware devices through interrupts, or from the running program/CPU through traps and exceptions..

Hardware interrupt

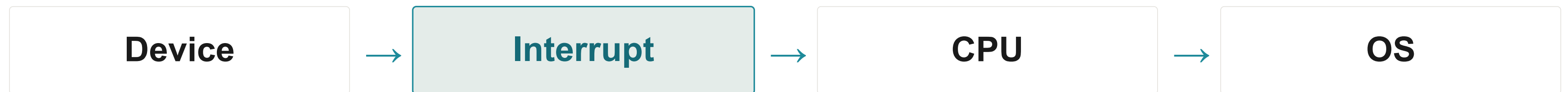
Raised by a **device through the system bus** — a completed transfer, input ready, or an error condition.

Software interrupt

Generated **from inside execution** . Two kinds, and the distinction matters: a **trap** and an **exception**.

Interrupt = External

something outside the CPU execution needs attention.



Example: the disk finishes reading → raises an interrupt.

MEMORY RULE

Something **outside** the CPU's execution needs attention.

Analogy — someone **outside** knocks on the door: external and unexpected.

System Call = Request

Not an event like an interrupt or exception — a **request for a service**.

“OS, please do something **privileged** for me.”

Examples

`open()` `read()` `write()` `fork()`

MEMORY RULE

WHAT service the program wants.

Analogy — the student asks, “**Can I go to the library?**” That is the actual request.

Trap = Controlled Entry into the Kernel

An **intentional, synchronous transfer** from a running program into the kernel.

TYPICAL FLOW



Analogy — a student **deliberately raises a hand** : the intentional way to get the teacher's attention.

Trap — Asking the OS for Help

Purposeful and synchronous: the running program **deliberately transfers control** to the OS.

SYSTEM CALL

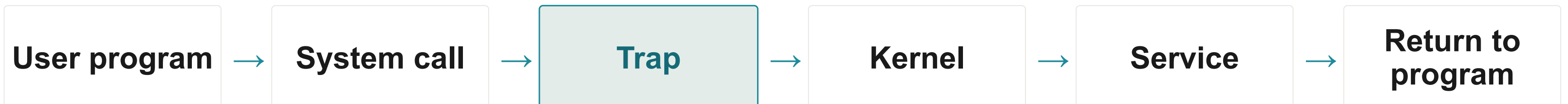
The **request**.

“**What** service do I want?”

TRAP

The **mechanism** that enters the kernel.

“**How** do I enter the kernel to get it?”



Asking the OS to:

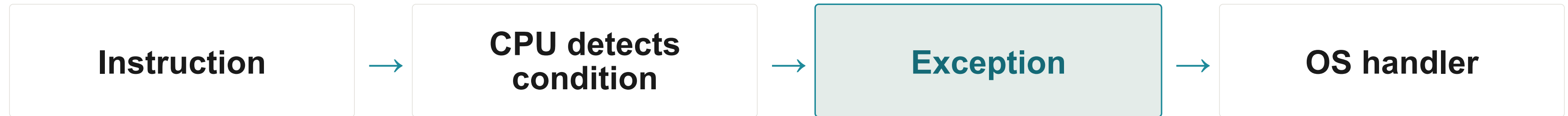
- open a file
- allocate memory
- read / write
- create a process

KEY IDEA

Trap = “I need the OS to do something for me.”

Analogy — a student **intentionally raising their hand** to ask the teacher for help.

Exception = Caused by the Current Instruction



Examples

- divide by zero
- page fault
- invalid memory access
- illegal instruction

MEMORY RULE

Happened **because of** the current instruction.

Not always an error — a page fault can be normal.

Analogy — a student tries $10 \div 0$: something goes wrong mid-operation.

Trap and Exception

TRAP

Purposeful and intentional.

Generated **by a program** to request an OS service.

Example: a program executes a system call — open a file, allocate memory.

Controlled transfer to the OS — commonly used for system calls.

Analogy — Trap: a student intentionally raises a hand to ask for help; Exception: while solving a problem, the student encounters an invalid operation such as division by zero.

EXCEPTION

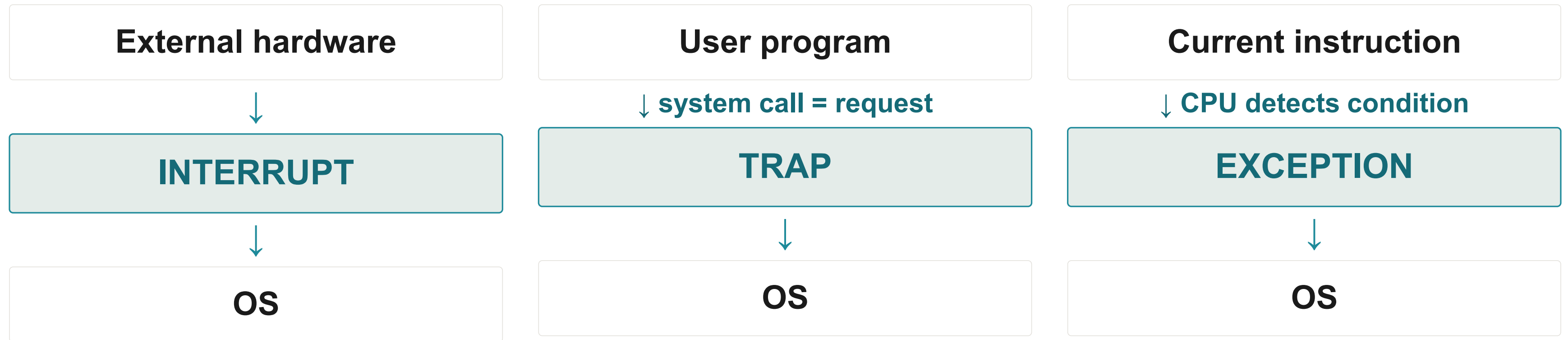
Generated automatically by the CPU because of an execution-time condition.

Synchronous. It is directly related to the current instruction..

Example: division by zero, invalid memory access, illegal instruction.

Requires handling: the OS determines whether execution can continue or must be terminated.

Three Paths into the OS



Different doors — **the same destination**: the OS takes control.

The Classroom

INTERRUPT

Someone **outside knocks** on the door.

TRAP

A student **deliberately raises a hand**.

SYSTEM CALL

The student asks: “**Can I go to the library?**”

EXCEPTION

Mid-problem, the student attempts **$10 \div 0$** .

One teacher (the OS), **four different reasons** to stop and respond.

One Line Each

Interrupt

external event

Trap

intentional entry to the OS

Exception

CPU-detected execution condition

System call

request for an OS service

In this course — “trap” = intentional entry, usually for a system call; “exception” = condition detected automatically during execution. Architectures vary.

Four Concepts, Four Lenses

CONCEPT	SOURCE	INTENT?	TIMING	PURPOSE	EXAMPLE
Interrupt	Hardware device	Usually no	Async	Device needs CPU	Keyboard, disk done
Trap	Running program	Usually yes	Sync	Enter the OS	Enter kernel for a service
Exception	Current instruction	Usually no	Sync	Condition during execution	Divide by zero, page fault
System call	User program	Yes	Sync	Privileged OS service	<code>read()</code> <code>write()</code> <code>open()</code>

Only the interrupt is asynchronous — the other three come from the running program.

Why Programs Need System Calls

A **system call** is the controlled way for a user program to ask the operating system to perform a task the program **cannot do directly**.

Why? Because programs run in **user mode** and are **restricted from directly accessing hardware or sensitive resources** — for **protection and security**.

So when a program needs something **privileged**, it asks the operating system through a system call — the **official, controlled entry point into the kernel**.

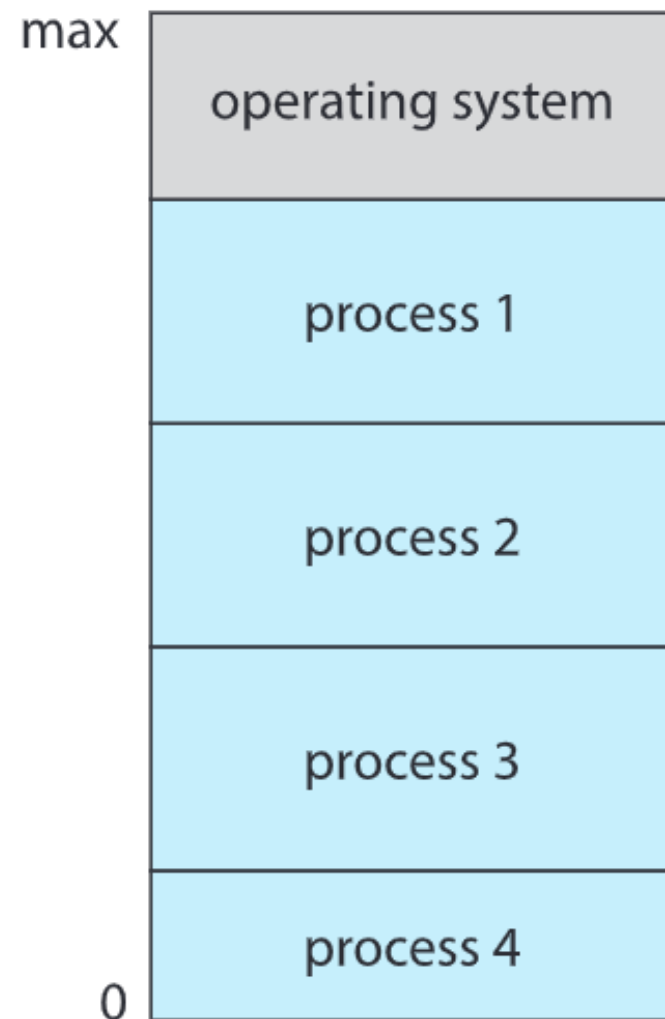
A program should not be allowed to directly:

- control the disk hardware,
- modify another process's memory,
- change page tables,
- disable interrupts,
- directly access protected kernel data.

Instead, it asks the OS:

- Please do this operation for me.

Multiprogramming - Keep the CPU busy as much as possible.



One chef serving several dishes.
Dish 1 goes into the oven for 10 minutes.
The chef does not stand beside the oven
doing nothing.



Dish 1 waiting in oven
→ chef works on Dish 2
When Dish 1 is ready, the oven timer signals the chef.

Keeping **multiple processes in memory** and letting the OS **switch among them**.

Goal: maximize CPU utilization — the CPU should never sit idle if another process is ready.

1. The OS picks one process and runs it.
2. If it needs I/O, the CPU would normally wait.
3. Instead, the OS **switches to another process**.
4. Once the I/O finishes, the first process can resume.

Multiprogramming Is Not Parallel Processing

Multiprogramming

One CPU core , several processes resident in memory. The OS **interleaves** them: only **one instruction stream executes at any instant** .

Parallel processing

Multiple CPUs or cores. Instruction streams genuinely execute **at the same time** , one per core.

Multiprogramming **removes idle waiting**; parallelism **adds simultaneous execution** . A modern system does **both**.

Multitasking = Time-Sharing

The OS switches the CPU among processes **very quickly** — so users feel many programs run **at the same time**.

An **extension** of multiprogramming.

Multiprogramming vs. Multitasking

MULTIPROGRAMMING

Keep the **CPU busy**.

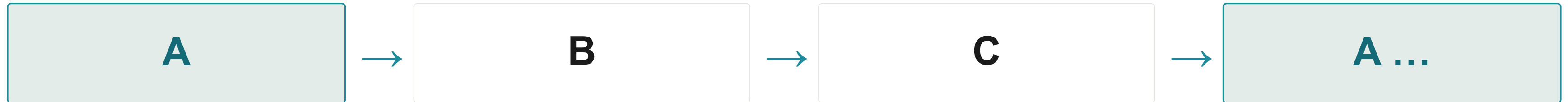
One process **waits for I/O** → run another.

MULTITASKING

Keep the system **responsive**.

Each process gets a **short time slice** → switch.

Short Slices, Fast Switching



User sees: “**Chrome, Word, music, and Teams all running together.**”

Single core: one process at a time — an illusion.

Multicore: some processes truly parallel.

Humans Are Extremely Slow

Interactive programs wait for:

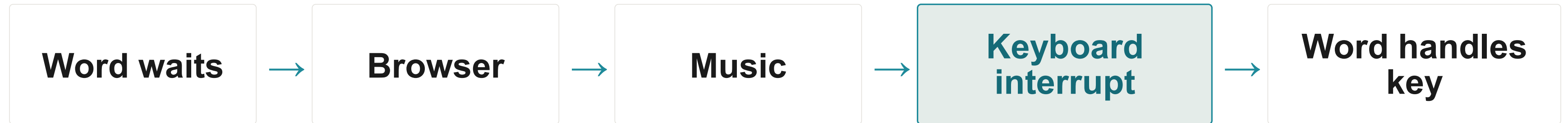
- typing
- mouse moves & clicks
- reading the screen
- thinking

Typing 7 characters / second

≈ 140 ms per key

For a CPU, that's **millions or billions** of operations.

While Word Waits for a Key



The CPU **never sits idle** waiting for the human.

Response Time

“How quickly does the computer respond **after I act?**”

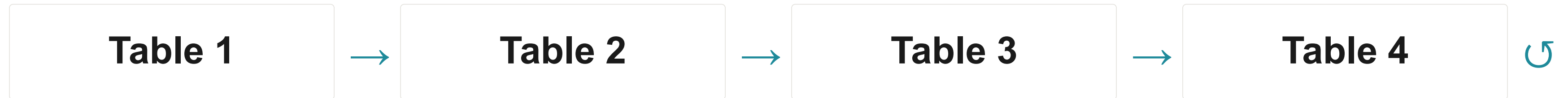
Click a button → **how long** until something appears?

Press a key → **how long** until the character shows?

Multiprogramming: CPU utilization

Multitasking: responsiveness + utilization

One Waiter, Four Tables



The waiter **doesn't stay** at Table 1 until the meal is finished.

Switching often → **every table feels attended to.**

Remember This

Multiprogramming one waits → run another

Multitasking switch rapidly → all stay responsive

Fast switching + short time slices = **illusion of simultaneity**

A **process** is a program in execution.

A **program** by itself is just **passive code stored on disk** .

Once **loaded into memory and executed** , it becomes an **active entity** — a process.

- A **program** is passive. It is simply a set of instructions stored somewhere, usually on an SSD or disk.
- chrome.exe stored on disk
Microsoft Word stored on disk
a Python file stored on disk

A **process** includes more than just the program code.
current CPU state and registers –
program counter – stack – heap – data -
open files and other resources.

- opening several instances of the same application can produce multiple processes.

What Multitasking Demands of the OS

Memory management

Keep **several processes in memory at once** — tracked, allocated, and **protected from one another** .

CPU scheduling

When several processes are **ready**, choose **which one runs next** .

Virtual memory

Run programs **larger than physical memory** , by executing a process that is **not completely in memory**.

Needed pages → RAM
other pages → SSD / secondary storage (Page Fault)

On top of these: file-system management on secondary storage, storage management, **protection of resources** , process **synchronization and communication**, and **freedom from deadlock**.

Physical and Logical Memory

Physical memory

- The **actual hardware memory (RAM)** installed in the machine — limited in size, 8 GB or 16 GB.
- Managed directly by the OS and the hardware **MMU** (Performs address translation and protection checks in hardware)
- It stores the OS, processes, and data during execution.

The **real bookshelf** where the books are placed.

Logical memory

- Memory **as seen by a process** : each one “thinks” it has a **large, continuous space** from address 0 up to some maximum.
- Actually **mapped onto physical memory** by the OS and MMU, and **may be larger than RAM** through paging and swapping.

The **catalog** a student orders from — a huge library, on a smaller shelf.

ABSTRACTION

Programmers need not worry about **real memory size or locations** .

PROTECTION

Each process gets its own logical memory and **cannot reach another's** .

FLEXIBILITY

Programs **larger than RAM** can still run.

Physical & Logical Memory

Why the OS separates the real RAM from what a process sees

The Real RAM Chips

The **actual hardware** holding running instructions and data.

8 GB

16 GB

32 GB

OS manages RAM allocation.

MMU translates addresses and checks protection.

Analogy — the **real bookshelf**, with a fixed size.

The Process's View of Memory

Each process sees its **own address space**: $0 \rightarrow \text{max}$.

The process doesn't need to know:

- where its data sits in RAM
- which RAM frames are used
- what is on secondary storage

Logical address

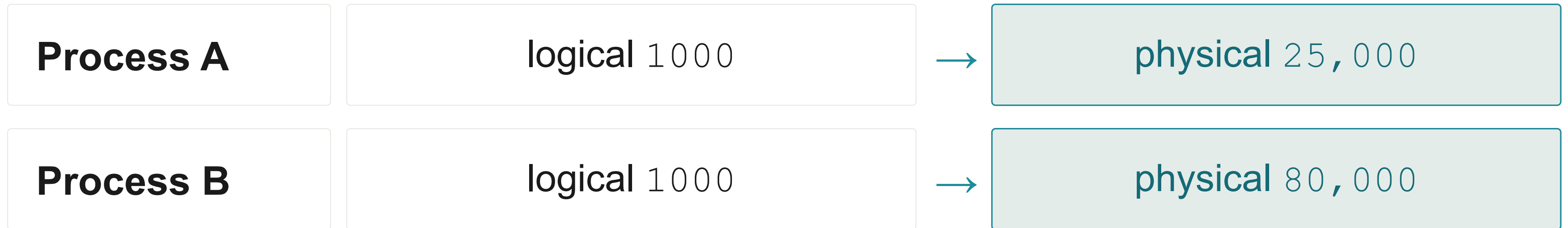


MMU



Physical address

Same Logical Address, Different RAM



Each process gets its **own independent memory space**.

Three Major Benefits

01

Abstraction

Hides the **complexity of physical RAM** .

02

Protection

Process A **can't access** Process B's memory.

03

Flexibility

Run **without all pages** in RAM.

The **MMU** and **OS** enforce isolation — improving security.

Bigger Than RAM

Process address space

10 GB

Available physical RAM

4 GB

WHEN A MISSING PAGE IS NEEDED

CPU accesses page



Page fault



OS loads page



**Execution
continues**

A process **needn't** be fully in RAM to execute.

The Library Bookshelf

PHYSICAL MEMORY

The **actual bookshelf** — only so many books fit.

LOGICAL MEMORY

The **catalog** — a much larger collection.

Book not on the shelf? The **librarian fetches it** — that's a **page fault**.

Remember This

Physical memory

real RAM

Logical memory

each process's view

MMU + OS

map one to the other

The logical address space **may exceed available RAM** through virtual memory, such as **paging**.

Why Modes Are Needed

The OS and user programs share the same hardware. We must **protect the OS from buggy or malicious user code** — so the hardware provides **different modes of execution**.

```
mode bit = 1
```

User mode

Normal applications (browser – Word - Python program - media player) run here, with **limited privileges**.

```
mode bit = 0
```

Kernel mode

Also **supervisor, system, or privileged mode** : the OS runs here, with **full access to the hardware**.

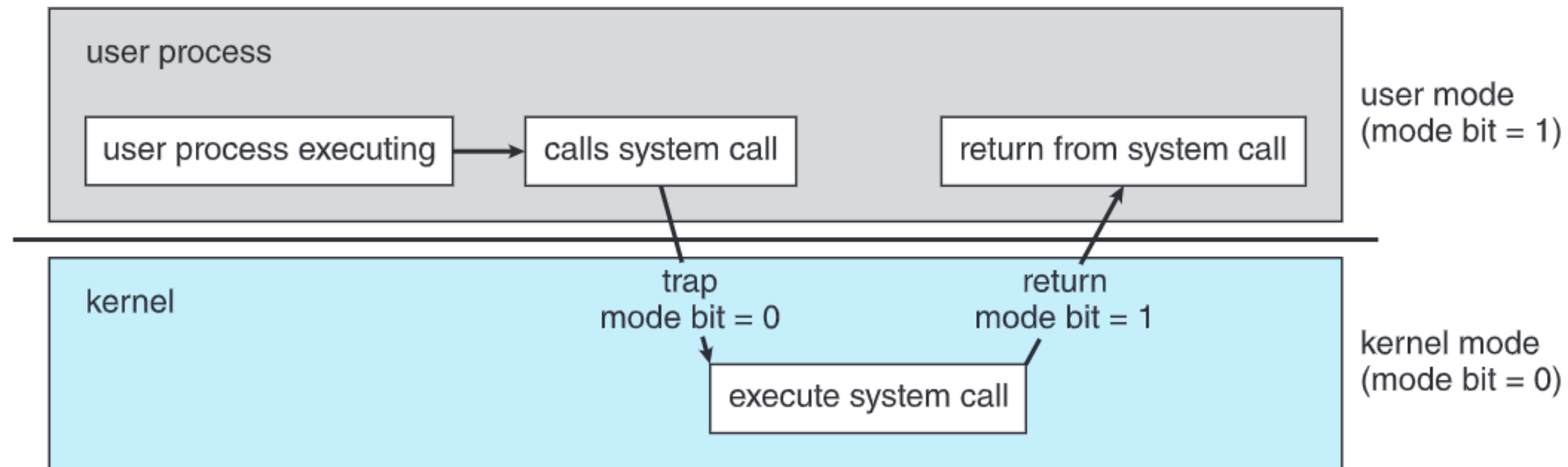
1. At boot time the system starts in **kernel mode**.

2. The OS loads, then starts applications in **user mode**.

3. An interrupt, trap, or system call **switches to kernel mode** .

4. Before returning to user code, it **switches back**.

Transition from User to Kernel Mode



The user process calls a **system call** ; the **trap** sets the **mode bit to 0** .

The **kernel** executes the **system call** with **full privileges**.

On return the **mode bit goes back to 1** and the **user process resumes** .

Privileged Instructions

(تعليمات مميزة) - تنفذ فقط في نمط النواة

Some instructions are **dangerous**. They are allowed **only in kernel mode** ; attempted in user mode, the hardware treats them as **illegal** and **raises a trap to the OS**.

Switching to kernel mode

I/O control

Timer settings

Interrupt management

Dual-mode operation protects the OS from **errant users** — and errant users **from one another** .

Think of protection as traffic rules.

User mode

Normal drivers: limited actions, must follow the rules.

Kernel mode

Police cars: can do anything, and **control the traffic** .

Mode bit

The **badge** showing **who is in control right now** .

Illegal attempt

A driver acting like a police car is **caught and redirected** — a **trap to the OS**.

Beyond Two Modes

Intel x86

Four protection rings: ring 0 is kernel, ring 3 is user. Rings 1 and 2 are **rarely used**.

ARMv8

Seven modes of operation.

Virtualization

An **extra mode** for the **virtual machine monitor (VMM)** , sitting between user and kernel: **more privileges than user processes, fewer than the kernel** .

Life cycle: control starts in the **OS in kernel mode**, passes to an **application in user mode** , and returns to the OS via an **interrupt, a trap, or a system call**.

How a System Call Is Executed

The **official way** for a user program to request an OS service. Typically implemented as a **trap instruction** — a software interrupt through the **interrupt vector**.

01

The trap causes a **switch to kernel mode** ; control passes to a **service routine** in the OS.

02

Parameters say what service is requested and what data is needed — passed via **registers, the stack, or memory** .

03

The kernel **checks that the parameters are correct and legal** , then executes the request.

04

Control returns to **user mode**, at the **instruction following the system call** .

Error Handling

If a user program attempts an **illegal memory access** or an **illegal instruction**, the hardware **raises a trap**.

Illegal operation → CPU detects it → control transfers to OS → OS handles the fault → process may be terminated

The OS **catches it, terminates the program abnormally**, and may write a **memory dump** to a file so the programmer can **examine it, correct the fault, and restart the program**.

The **same code** handles both **hardware-detected faults** and **user-requested abnormal termination** .

We must prevent a user program from hogging (احتكار) the CPU.

An **infinite loop** must not be able to keep the CPU forever: the operating system **must always have a way to regain control** . The solution is a **timer**.

How the Timer Works

1. The OS **sets the timer** before giving control to a user program.

2. The timer **counts down** while the program runs.

3. When it reaches **0**, it **generates an interrupt**.

4. The CPU **switches back to the OS**, in kernel mode.

5. The OS decides: **end the program as a fatal error**, or **reset the timer** and allow it more time.

TYPES OF TIMER

Fixed period — an interrupt every **1/60 second**, for example.

Variable period — the OS chooses the interval, implemented with a **fixed-rate clock plus a counter**.

```
10-bit counter + 1 ms clock → 1 ms to 1,024 ms
```

Clearly, instructions that **modify the content of the timer are privileged**.

Linux Timers: HZ and jiffies

HZ

The kernel configuration parameter that defines the **frequency of timer interrupts**.

HZ = 250 means **250 interrupts per second** — one every **4 ms**, since $1000 \div 250 = 4$.

Its value depends on **kernel configuration**, **hardware type**, and **CPU architecture**.

jiffies

A **global counter** in the Linux kernel: how many **timer ticks** have occurred **since the system booted**.

One jiffy is the time between two timer interrupts — the duration of a **single tick** of the system timer.

$\text{HZ} = 250 \rightarrow 1 \text{ jiffy} = 4 \text{ ms}$ $\text{HZ} = 1000 \rightarrow 1 \text{ jiffy} = 1 \text{ ms}$

What to Take from This Lecture

- 01 A **bootstrap program in firmware** initializes the hardware and loads the kernel; system programs become **daemons**, and on Linux **systemd** starts them.
- 02 A **trap is intentional** — the way into the OS; an **exception is an error** the OS must handle.
- 03 **Multiprogramming** keeps the CPU busy; **multitasking** switches fast enough to feel simultaneous — and **neither is parallel processing**.
- 04 The **mode bit** separates user from kernel mode; **privileged instructions** run only in the kernel, and **logical memory** keeps processes apart.
- 05 A **timer interrupt** guarantees the OS regains control. **Next:** Section 1.5, Resource Management.